BEST AVAILABLE IMAGES

Defective images within this document are accurate representations of the original documents submitted by the applicant.

Defects in the images may include (but are not limited to):

- BLACK BORDERS
- TEXT CUT OFF AT TOP, BOTTOM OR SIDES
- FADED TEXT
- ILLEGIBLE TEXT
- SKEWED/SLANTED IMAGES
- COLORED PHOTOS
- BLACK OR VERY BLACK AND WHITE DARK PHOTOS
- GRAY SCALE DOCUMENTS

IMAGES ARE BEST AVAILABLE COPY.

As rescanning documents *will not* correct images, please do not report the images to the Image Problem Mailbox.

Look-Ahead Processors

ROBERT M. KELLER

Department of Electrical Engineering, Princeton University, Princeton, New Jersey 08540

Methods of achieving look-ahead in processing units are discussed. An optimality criterion is proposed, and several schemes are compared against the optimum under varying assumptions. These schemes include existing and proposed machine organizations, and theoretical treatments not mentioned before in this context. The problems of eliminating associative searches in the processor control and the handling of loop-forming decisions are also considered. The inherent limitations of such processors are discussed. Finally, a number of enhancements to look-ahead processors is qualitatively surveyed.

Keywords and Phrases: asynchronous computation, computer architecture, computer organization, look-ahead, parallelism, pipelining, schemata

CR Categories: 5.24, 5.5, 6.32, 6.33

INTRODUCTION

Arithmetic and logical processors in computers of the "second generation" and earlier tended to be unsophisticated insofar as their highly serial nature of instruction execution was concerned. Furthermore, the bottleneck created by a relatively slow core memory with a single-access port made the problem of enhancing the processor's speed uninteresting. With the advent of such techniques as multiple-port interleaved memories, semiconductor memories, and the use of more fast, local registers (either programmable or cache), we have the capability of transmitting operands and results between processors and memory at a much faster rate. The ability to provide a corresponding rate of instruction execution, then, depends on the speed of the processor.

Techniques for enhancing the speed of a processor by "look-ahead" are examined in this paper. The term look-ahead derives from a class of schemes in which programs for the processor are specified in a conventional, serial manner; however, the processor can look ahead during execution and execute instructions out of sequence, provided no logical inconsistencies arise as a result of doing so. The advantage of look-ahead is that several instructions can be executed concurrently, assuming the processor has sufficient capabilities. Designs of specific look-ahead processors have been presented in [AST, Th, To].

Using the diagrams in Figures 1 and 2, this paper will model a computer with a look-ahead processor. We are concerned mainly with the processor here, and not the overall system. Note that the processor contains a number of *local registers* for the storage of data, a number of *function units* for operating on this data, and an instruction buffer, or *window*, for storage of instructions. We use the term "window" because the instruction buffer can be viewed as looking onto a small segment of the program in execution.

We assume that the programs to be exe-

CONTENTS

cuted specify *serial execution*. That is, the correct semantics of execution are defined by the execution of one instruction at a time, in the order specified. It is the task of the look-ahead processor to determine which instructions can be executed concurrently without changing the semantics.

Once a processor has determined which instructions can be executed concurrently, it must assign them to the available physical function units. Hence, the processor performs two main tasks:

1) *detection of parallelism*—determining which instructions may be executed concurrently, and
2) *scheduling*—assigning concurrently executable instructions to function units.

We shall see that the detection of parallelism is largely a machine-independent task, whereas scheduling—at least optimal scheduling—must generally take into account the specific number of function units available.

Some additional tasks which might also be performed by such a processor are:

3) register assignment and renaming, and
4) modifying code according to algebraic identities.

We shall not be concerned with these tasks initially, but will comment on them later.

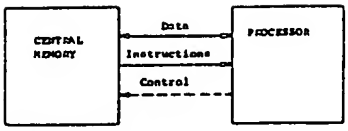


FIGURE 1. Computer model.

The emphasis here will be on schemes that approach optimality in the detection of parallelism. Many definitions of optimality are possible, depending on the choice made among a set of possible "ground rules." Several possibilities will be mentioned here, and we shall give techniques for approaching optimal look-ahead for some of these.

In discussing optimality, there are two major categories:

1) Global optimality—optimality with respect to the execution of entire programs; and
2) Local optimality—optimality with respect to the contents of the window only.

Discussing global optimality for general programs appears to be an extremely difficult problem. Indeed, it is even difficult to say what we mean by "optimal" in this case, since the class of all possible programs is extremely large. In contrast, by suitably restricting our model we can say some things about local optimality.

A further division occurs within the context of local optimality. There appear to be two types: *static* and *dynamic*. This distinction arises from the fact that the window contents may be constantly changing. That is, when one instruction in the window has been completely executed, it may be "retired" and a new instruction brought in. Then what started out as an optimal strategy before the new instruction was added may end up being nonoptimal. This is what we mean by "dynamic."

As with global optimality, local optimality in the dynamic case is difficult to define. We therefore restrict ourselves to the static case at present. Thus, we are interested in optimal behavior with respect to fixed

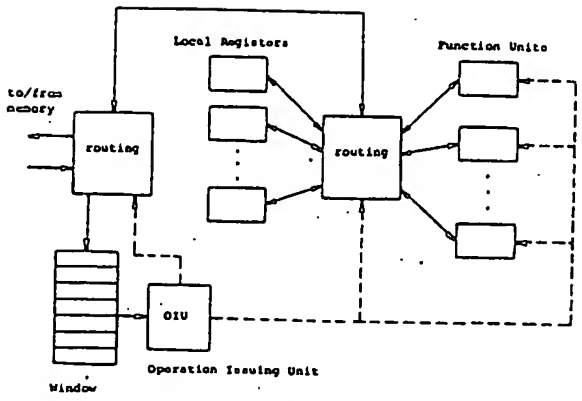


FIGURE 2. Processor detail.

window contents, as an approximation to a true local optimum. The approximation will be best in cases where the rate of change of the window is slow in comparison to the execution rate. This occurs, for example, if a program loop is entirely contained within the window and the loop executes a substantial number of iterations.

We shall now specify the model in more detail. Assume the window contents represent a segment of the program being executed, with elements of that segment being statements of the form (here h is to be read as a subscript):

s: $i \leftarrow F_h(j, k)$
or s: if $G_h(j)$ then goto s'
or s: goto s'
or s: exit

Here s and s' represent instruction labels, which are implicitly associated with the instructions; i, j, and k represent registers local to the processor; F_h represents one of a set of functions (such as "add," "multiply," etc. We assume that each function has two arguments for simplicity. It will become apparent that this causes little loss of generality.); G_h represents one of a set of test predicates (such as "compare to zero"); and "exit" signifies to the control that the next statement is not in the window (not the end of the program). Hence, "exit" indicates to the control that further instructions must be fetched.

We will not concern ourselves with the mechanics of fetching instructions or operands from the central memory, but will assume that this is handled by a mechanism external to that of the look-ahead. It suffices to say that this fetching is heavily overlapped with instruction execution. Multiple-memory modules and interleaving may be used to achieve a sufficiently high rate of instruction flow. The fetching of an operand for register i can be represented as an assignment $i \leftarrow F_h$, where F_h takes no arguments, and the storage of an operand can be represented by assigning one or more local registers to play the role of a storage buffer.

THEORETICAL BASIS

We present here a theoretical basis for the construction of look-ahead schemes, as well as certain assumptions that are made for convenience in comparing various schemes with respect to optimality.

We will call a statement with specific indices h, i, j, k an *operation*. Each statement, then, is called an *instance* of an operation. Operations corresponding to the "if" instruction discussed in the introductory section will be called *decisions*. In this section, we will be primarily concerned with the

decision-free case, in which decisions will not play a role in look-ahead.

It is assumed that the reader understands what is meant by *sequential execution* of a program or a program segment. We therefore present the following assumptions without further explanation.

Assumption 1: We are interested in possibly-parallel executions of segments that are *equivalent* to sequential execution, in the sense that the sequences of register contents are the same as they would be in the sequential case.

One consequence of this assumption is that if the issuance of instructions were suddenly stopped, for example by a program interrupt, then the resulting state of the machine would be invariant after all statements in the segment were completely executed.

The following assumptions (2 and 3) need not hold. Their purpose is to establish ground rules for comparing various schemes.

Assumption 2: We assume that no nontrivial relations (such as function equality) hold between two functions or tests of different indices.

This assumption is conservative, and it simply means certain algebraic reductions that might otherwise preserve equivalence are not allowed.

Assumption 3: We assume there are no inessential instructions; that is, no two instructions compute or test identical values.

Like its predecessor, this assumption is conservative. It is the informal equivalent of the "repetition-free" assumption in [KM2, Ke]. Its purpose is to prevent consideration of the assignment of infeasible program-optimization tasks to the processor. We may assume that preprocessing has already removed any inessential instructions.

Suppose b is an operation. We define the *domain registers* $D(b)$ and *range registers* $R(b)$ as follows: If b is an instance of

$$i \leftarrow F_A(j, k)$$

then $D(b) = \{j, k\}$ and $R(b) = \{i\}$. If b is an instance of

$$\text{if } G_A(j) \text{ then goto } s'$$

then $D(b) = \{j\}$ and $R(b) = \emptyset$. If b and c are two different operations, then we write *conflict* (b, c) if, and only if, either

$$R(b) \cap D(c) \neq \emptyset$$
$$\text{or } R(c) \cap D(b) \neq \emptyset$$
$$\text{or } R(b) \cap R(c) \neq \emptyset.$$

Otherwise, we write *no-conflict*(b, c).

It is a fact that ~~that~~ *no-conflict* (b, c) is a *sufficient* condition for a pair of operations b, c to be executed concurrently, or in either order, asynchronously (i.e., without regard to timing), and still remain equivalent to sequential execution in the sense described in Assumption 1. This is intuitively clear, but has been observed in [B], and given formal treatment in [Ke, KM2].

To see why this condition is generally *necessary*, if b and c are executed concurrently and $R(b) \cap D(c) \neq \emptyset$, then some value that c fetches is generally dependent on whether b has stored anything into a register common to $R(b)$ and $D(c)$. Similarly, if $R(b) \cap R(c) \neq \emptyset$, then the net result is dependent on whether b or c was the last to store something into a register common to both $R(b)$ and $R(c)$. The condition would not be necessary if the value stored by b happens to be the same as that fetched or stored (respectively) by c. We can see, however, that either case would constitute a violation of Assumption 3. Hence, we will assume that this condition is both necessary and sufficient for the concurrent execution of two operations.

In discussing techniques for detecting parallelism, we are therefore interested in schemes that preserve the order of operations b, c in which *conflict* (b, c) is the condition. We will also assume for now that b and c are not decisions, for to do otherwise would require some method of specifying precisely what it means to interchange them.

Combined with the serial ordering of instructions, the conflict relation gives rise to a *precedence relation*. We say *precedes* (s_i, s_j) if, in any execution, s_i must be executed before s_j. The precedence relation may be determined from both the conflict relation and the given serial ordering as follows: assume that $s_1, s_2, \cdots, s_k$ is the serial ordering specified by the program.

we write

b, c) is a
erations
in either
t regard
alent to
cribed in
ear, but
a formal

enerally
urrently
e value
lent on
register
arly, if
it is de-
last to
mon to
would
d by b
ched or
an see,
titute a
we will
cessary
ecution

tecting
sted in
opera-
condi-
that b
erwise
cifying
change

ing of
rise to
(s_i, s_j)
ecuted
may
t rela-
ng as
is the
gram.

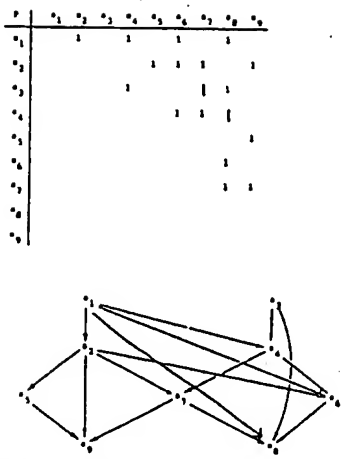


FIGURE 3. The P relation for Example I and its graph. (The conflict graph is obtained by making each arrow bidirectional.)

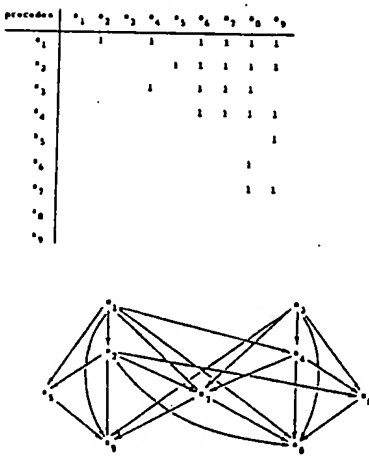


FIGURE 4. The "precedes" relation for Example I and its graph.

Let $P(s_i, s_j)$ be true if, and only if, $i < j$ and *conflict* (s_i, s_j). Then *precedes* is the "transitive closure" of P; that is, *precedes* (s_i, s_j) if, and only if, there is a sequence $i = i_1 < i_2 < \cdots < i_r = j$ such that

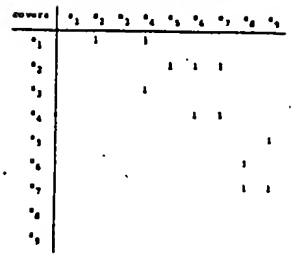


FIGURE 5. The "covers" relation for Example I and its graph.

conflict (s_{i_1}, s_{i_2}), *conflict* (s_{i_2}, s_{i_3}), $\cdots$, *conflict* $(s_{i_{r-1}}, s_{i_r})$.

Reasonably efficient algorithms for computing the transitive closure of a relation are known; cf. [War].

Actually, another relation is more important than *precedes* for our purposes. This relation, which we call *covers*, is the "cover" of the relation *precedes*. (It is the cover of P as well.) That is, *covers* is the smallest relation whose transitive closure is *precedes*. This condition is obtained by simply removing redundant arcs which are implied by transitivity. An algorithm for computing "covers" is discussed in [AGU], wherein the "covers" relation is referred to as the "transitive reduction." Studying the example in Figures 3, 4, and 5 should make the meaning of these relations and their computation reasonably clear. For a formal analysis, see [Ke].

As a further basis for comparing different look-ahead schemes, we shall assume that instructions are examined *sequentially* in the generation of operations. In other words, we hypothesize a unit, called an *operation issuing unit* (OIU), which examines the contents of the window one instruction at a

time and which either issues the corresponding operation to a function unit, or decides to defer the issuance for some reason. For now, we assume that an operation is *issued* by transferring the *indices* of the registers involved to the appropriate function unit, which then operates on the values in the registers indexed.

The issuance of operations proceeds concurrently with the entrance and exit of instructions in the window. It is possible to examine instructions in parallel; however, we contend that sufficient speed can be achieved by sequential examination. Furthermore, parallel examination of instructions appears to increase, unduly, the complexity of the control.

If the issuance of an operation corresponding to the instruction being examined is deferred, we say that the operation is *pending*. If the operation has been successfully issued, we say that it is *being executed* until the time it is *completed*.

To summarize this section, we may state the following:

Principle of Optimality for operation issue (decision-free case): Whenever c is an operation corresponding to an instruction in the window, and there is no operation b which is either being executed or is pending execution such that *conflict*(b, c), then operation c should be issued.

ELEMENTARY SCHEMES

We shall now examine some schemes for detecting parallelism by using the principle of optimality stated in the previous section. For initial simplicity, we assume that the window does not contain any decisions; rather, detection of a decision is accomplished external to the window, and it causes the transfer of instructions into the window to halt until the decision is resolved.

The first scheme will be called the *simple indicator scheme*. As we shall see, this method is closely related to a scheme discussed in [To], but we have elaborated it slightly for the sake of explanation. With each register i is associated an *indicator register*, W_i, which holds an encoded value from the set $\{1, 0, -1, -2, \cdots, -N\}$ where N is the maximum number of concurrently-executable operations. If $W_i = 1$, then an operation b is in progress such that $i \in R(b)$. If $W_i = -m$, then there are m operations b in progress with $i \in D(b)$. If an operation b is in progress with $i \in D(b)$ and $i \in R(b)$, then it will be the case that $W_i = 1$ by convention. In addition, a one-bit register B_k is associated with each function unit F_k. $B_k = 1$ if an operation using function unit F_k is in progress, and $B_k = 0$ otherwise.

The operation-issuing unit (OIU) examines the instructions in the window sequentially, and if the necessary conditions are satisfied, it issues the instruction to the appropriate function unit. If the instruction is

$$i \leftarrow F_k(j, k)$$

then the conditions which must be satisfied are

$$W_i = 0$$
$$W_j \leq 0$$
$$W_k \leq 0$$
$$B_k = 0$$

Once the operation is issued, and before the next instruction is examined, the OIU sets $W_i = B_k = 1$, and sets $W_j \leftarrow W_j - 1$, and $W_k \leftarrow W_k - 1$ (unless j or $k = i$, in which case W_i is set to 1 as discussed above). When the operation is completely executed, B_k and W_i are set to 0 and $W_j \leftarrow W_j + 1$ and $W_k \leftarrow W_k + 1$ (unless j or $k = i$, as in the previous case).

The completion of the operation is determined in the synchronous case by a specific elapsed time, or, in the asynchronous case, by notification by the function unit itself. At this point in the discussion this detail is unimportant.

Example 1: We consider the following window contents as part of a running example:

	instruction	*operation*
a_1 :	$1 \leftarrow F_1(2, 3)$	(a)
a_2 :	$5 \leftarrow F_3(1, 2)$	(b)
a_3 :	$4 \leftarrow F_2(2, 2)$	(c)
a_4 :	$3 \leftarrow F_1(1, 4)$	(d)
a_5 :	$6 \leftarrow F_1(5, 6)$	(e)
a_6 :	$1 \leftarrow F_1(2, 3)$	(a)
a_7 :	$4 \leftarrow F_2(2, 5)$	(g)
a_8 :	$3 \leftarrow F_1(1, 4)$	(d)
a_9 :	$5 \leftarrow F_3(5, 5)$	(f)

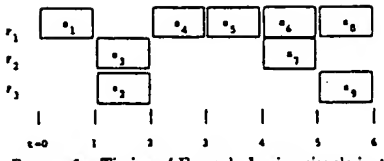


FIGURE 6. Timing of Example 1 using simple indicator scheme and one F_1 unit, assuming unit execution times for all operations.

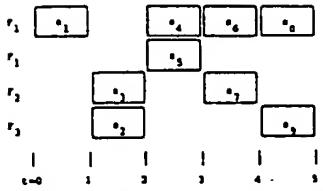


FIGURE 7. Timing using simple indicator scheme and two F_1 units.

The operation names shown here will not play a role in this discussion until we treat the problem of optimal schemes without associative search, in the section so titled.

In Figure 6, we illustrate a possible timing diagram for the simple indicator scheme with one F_1 unit, which each function is assumed to require one unit of time. We emphasize that this assumption is made for the purpose of illustration only. We also assume that the time required for actual scanning is negligible. A description of the scan in this case is as follows: At $t = 0$, s_1 starts. Since s_2 depends on s_1, s_2 is not issued and the scan stops. At $t = 1$, s_1 completes and s_2 starts. The scan then continues and s_3 starts. As s_4 depends on s_3, the scan stops. At $t = 2$, s_4 starts, but s_5 requires the same function unit as s_4, so s_5 stops the scan. At $t = 3$, s_4 starts, etc. The time required to completely execute the window contents is 6.

In Figure 7 we illustrate the timing diagram for the simple indicator scheme using two F_1 units. The scan in this case is similar to the previous one, except that at $t = 3/2$ both s_4 and s_5 can start. This has the effect of reducing the required time to 5 units.

The reader will note that the use of indicator registers in this way preserves the order of operations b followed by c whenever *conflict*(b, c). However, this use has the disadvantage that a wait for the satisfaction of a condition by an indicator will cause the issuance of instructions to halt temporarily, even though some successive instructions in the window might be issuable as operations. Hence, this method cannot be considered optimal.

Note that such "wait conditions" may result because of register conflicts, or because of the unavailability of the function unit. A scheme to partially nullify the latter constraint involves the introduction of *virtual function units*. (A similar concept is that of *reservation stations* [To].)

A virtual function unit is used to represent an operation which could be in progress, but which might not be due to the unavailability of a real function unit. We may think of such operations as forming a queue, which is served by the real function unit when it becomes available. This also seems to be a convenient way of organizing the allocation of several function units of the same type.

The advantage of the virtual function unit method is that it does not allow the issuance of operations to be impeded by busy function units, but only by register conflicts. Of course in practice, even the number of virtual functions will be bounded by hardware considerations, and a counter that indicates the number of units available would be used, halting the issuance of operations when the count becomes zero.

Figure 8 illustrates the same example, except that there is one F_1 unit and two virtual F_1 units. The scan passes to s_7, even though s_6 is waiting. Although the time required here is the same as in Figure 6, note that in Figure 8, F_2 finishes earlier. This could be exploited, were there more instructions to follow.

THE EFFECT OF BUFFERING

In many cases of practical interest, the actual function is computed from *buffers* for the domain registers, rather than from the registers themselves. Indeed, rather than having the issuance of an operation transfer the *indices* of the domain registers to the function unit, the *values* in the domain regis-

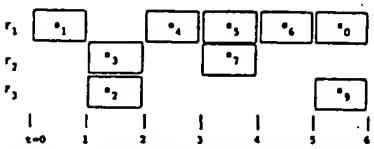


FIGURE 8. Timing using simple indicator scheme with one F_1 unit and two virtual F_1 units.

ters can be transferred. This simplifies the design, with a small increase in time for execution due to the added buffering. However, it is likely that the function unit will be implemented with internal buffers, whether or not advantage is taken of this fact, as discussed in the following paragraphs.

The use of buffering has ramifications for increased concurrency. If b is the operation

$$i \leftarrow F_k(j, k)$$

and this operation is really executed as

$$\begin{aligned} x &\leftarrow j \\ y &\leftarrow k \\ i &\leftarrow F_k(x, y) \end{aligned}$$

where x and y are buffer registers that are distinct from all program-addressable registers, then we may relax the constraints on sequencing, provided the scan proceeds only after buffering has taken place. If c is an operation that follows b, then recall that we required

$$\begin{aligned} D(b) \cap R(c) &= \emptyset \\ D(c) \cap R(b) &= \emptyset \\ \text{and} \quad R(b) \cap R(c) &= \emptyset \end{aligned}$$

for concurrent execution of b and c. However, if we start c only after the buffering operations for b

$$x \leftarrow j, y \leftarrow k$$

have been done, we no longer need the constraint

$$D(b) \cap R(c) = \emptyset$$

because operation c cannot possibly affect the computed value of b. If buffering is done uniformly for all operations, then we see that the scheme can be simplified to require only a one-bit indicator for each register, with that bit indicating whether an operation which has the corresponding register in its

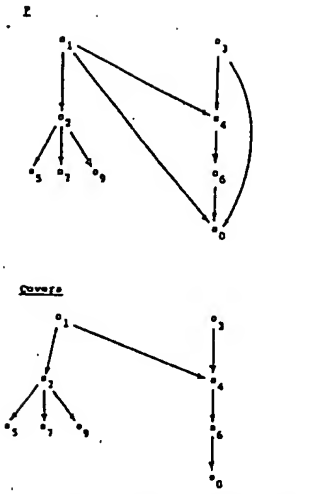


FIGURE 9. The P and "covers" relation for Example I with instantaneous domain and range buffering.

range is in progress. This is essentially the scheme described in [To] (pp. 28–29).

It is not difficult to show that buffering the *output* of a function unit can similarly remove the requirement $R(b) \cap R(c) = \emptyset$. That is, if $R(b)$ is buffered for use in the domain of any operation issued before c, then the range conflict requirement need not be of concern. Of course the requirement $R(b) \cap D(c) = \emptyset$ can never be removed, as this indicates a logical dependency between b and c.

In summary, if we can be sure that buffering occurs before the scan proceeds to the next instruction, then the P relation becomes $P(s_i, s_j)$ if, and only if, $i < j$ and $R(s_i) \cap D(s_j) \neq 0$.

Figure 9 shows the *covers* relation for Example I when both domain and range buffering are used, assuming that buffering effectively occurs instantaneously. The conflict relation, as defined in the section on "Theoretical Basis" is not meaningful in this case, since precedence now depends on the original ordering, as well as on which registers are used. We leave to the reader the task of

formulating a suitable analog of "conflict," as well as that of constructing the corresponding relation that explicitly shows buffering operations. (Note the presence of fewer arcs in Figure 9 than in Figure 5 (page 000), indicating the greater degree of concurrency possible.)

FORWARDING

To prevent the issuance of operations from being halted by register conflicts, an approach called *forwarding* can be used [To]. Our presentation is modified from that in [To] for clarity. (It is unfortunate that the description of the scheme in the cited paper is labeled with the implementation detail of a "common data bus." Indeed a bus, or any of several other possible means of interconnection, could be used to route the data between registers and function units. Our description ignores this detail.)

In this scheme, a register may contain a data value as before, or a specially-indicated "tag." A *tag* is simply an index of one of the virtual function units. If a register contains a tag, its proper contents have not yet been computed. The tag is, in fact, the index of the function unit from which the computed value will come.

We assume that each virtual function unit has domain buffers. Forwarding schemes without buffers will not differ much conceptually. With forwarding, if the OIU wishes to issue an operation b with $i \in D(b)$, but register i contains a tag, the unit *conditionally* issues the operation and passes the tag to the virtual function unit specified in the operation (or to the function unit if virtual function units are not used). This tag is kept in a buffer of the virtual function unit as an indication that the operation is not to proceed until the function unit corresponding to the tag completes its execution. Upon completion, the control checks the registers and buffers within all virtual function units, to see if any of their contents match the tag of the completing function unit. If a match occurs in the registers, the result of the completing operation is sent to the proper register. If a match occurs in the buffers, the result is *forwarded* to the conditionally-issued operation. When a conditionally-issued operation has all of the necessary operands, it is considered to be *issued* and may begin execution. The execution of part of Example I, using forwarding, is shown in Figure 10.

The primary advantage of the forwarding technique is that the examination of instructions does not stop simply because an instruction with a busy register is encountered. This means that if there are enough virtual function units, all potential concurrency will be detected without stopping the scan.

The main disadvantage with this implementation is that forwarding requires an associative search to match tags; this may either be time-consuming or require rather complex hardware implementations. The reader might observe that the need for associative searches could be overcome if we had a way of associating with each virtual function unit a list of registers to which its results are to be sent. Such a list might be implemented using a linked-list strategy, for example. However, there are some subtleties that limit the usability of this approach. The fact that a range register containing a tag may be overridden (e.g., in Figure 10) indicates that there will be difficulties in updating these lists. If the number of registers and buffers is small, it becomes feasible to use a bit vector to represent the register to which the results should be forwarded. Updating these bit vectors is substantially simpler than updating a linked-list. Other organizations which eliminate the associative search are discussed in the next section.

It is clear that the scheme using forwarding is optimal (provided there are enough function units) since an operation c is not executed if, and only if, there is an operation b which is either being executed or pending execution, such that *conflict*(b, c). That is, the *issuance* of operations is never halted because of a register conflict. Furthermore, forwarding incorporates domain and range buffering quite naturally. Because the forwarding scheme is combined with buffering, the indicators W_i are totally redundant, since $W_i = 0$ if, and only if, R_i does not contain a tag.

Figure 11 depicts the timing diagram for

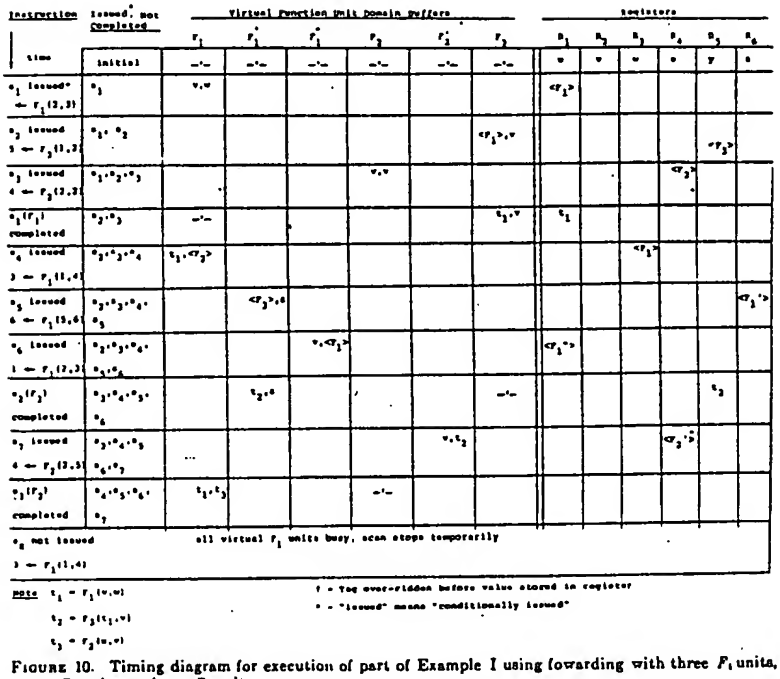


FIGURE 10. Timing diagram for execution of part of Example I using forwarding with three F_1 units, two F_2 units, and one F_3 unit.

Example I with virtual function units and forwarding using only one real F_1 unit. Figure 12 depicts the same example using two real F_1 units. As before, we assume for illustration that each operation requires one time unit. The optimality of this scheme is verified in Figure 12, as the absolute minimum time (4 units) is achieved, as indicated by the longest path in the graph in Figure 5 (page 181).

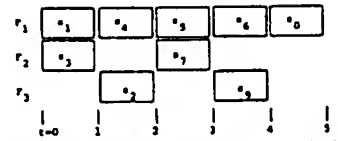
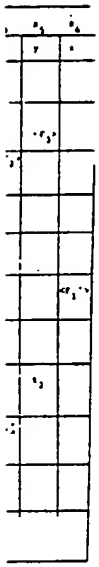


FIGURE 11. Timing for Example I with forwarding and one F_1 unit.

OPTIMAL SCHEMES WITHOUT ASSOCIATIVE SEARCH

In the previous section we mentioned the difficulties of eliminating the associative search that accompanies the forwarding scheme. Here we discuss other organizations that eliminate the search, and which were developed in more theoretical contexts. On a practical-implementation basis, these techniques may not be competitive with those discussed in the previous section. However, they provide a useful conceptual tool, and will be seen to fit the need nicely when we introduce the consideration of decisions. We assume, for initial simplicity, that domain buffering will not be used, then modify this assumption later on.

It was shown in [Ke] that optimality



ree F_1 units.

with for-

exts. On a
hese tech-
with those
However,
tool, and
/ when we
decisions.
, that do-
ien modify

optimality

could be achieved by a control scheme that employs first-in-first-out *queues* in the following manner. One queue is associated with each pair of conflicting operations. We will say that an operation *belongs* to a queue if it is one of the operations with which that queue is associated.

We think of the elements stored in a queue as being *tokens*, with a different token being associated with each operation. When the operation-issuing unit encounters an instruction, it simply places one token for the corresponding operation at the tail of each queue to which that operation belongs. Before an operation can begin, there must be a corresponding token at the head of each queue to which it belongs. When the operation completes, the tokens are removed. Each queue can be implemented as a linked-list.

It is easy to see that this scheme works because it preserves the necessary precedence between conflicting operations. It is optimal because it preserves only this precedence. An unfortunate property of this scheme is that the number of queues may be prohibitive, since, if there are m different binary functions and n different registers, we may have on the order of mn^2 different operations, and (it can be shown) on the order of mn^4 conflicts pairs.

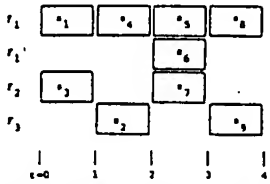


FIGURE 12. Timing for Example 1 with forwarding and two F_1 units.

To reduce the number of queues from one queue per conflict pair, we may use one queue for each of a set $C_1, C_2, \cdots, C_k$, where each C_i is a set of operations, provided that for every pair of operations b, c, *conflict*(b, c) if, and only if, there is an i such that $\{b, c\} \subseteq C_i$. As before, if $b \in C_i$, then we say that b *belongs* to the corresponding queue, and the description of the control mechanism holds as stated previously. This approach is illustrated in Figure 13.

We note that such a set of queues can always be obtained by associating one queue with each pair (b, i), where i is a register and b is an operation such that $i \in D(b)$. The operations that belong to this queue are b, together with those c, such that $i \in R(c)$. This reduces the maximum number of queues to mn.

We note that buffering may be used with the queueing scheme in a natural way. Each operation b is split into three parts, b_1, b_2, and b_3, such that b_1 corresponds to buffering one domain register, b_2 corresponds to buffering the other domain register, and b_3 corresponds to storing the result. The conflict relation can then be defined between parts of operations, rather than between the operations themselves, and queues can be defined accordingly.

A similar queueing scheme is discussed in [De]. Here there is one queue for each register. Any operation b for which $i \in D(b)$, or $i \in R(b)$ can appear as a token on the queue corresponding to i. However, the queue is not strictly first-in-first-out. Instead, if b and c are operations with $i \in D(b) \cup D(c)$, but $i \notin R(b) \cup R(c)$, then the tokens corresponding to b and c on the queue corresponding to i can be *interchanged* arbitrarily. However, if $i \in D(b) \cup R(b)$, and $i \in R(c)$,

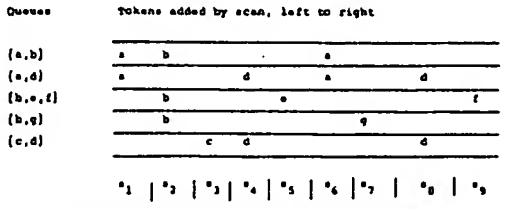


FIGURE 13. Queueing scheme applied to Example 1.

then the tokens corresponding to b and c cannot be interchanged.

The number of queues in this version may be substantially fewer than in the previous scheme, because there is only one queue per register. However, the fact that token interchanging can occur in a nondeterministic fashion casts doubt on the efficiency of such an implementation. Fortunately, this scheme can be rescued by using domain buffering and virtual function units, as described earlier. A modification of this type is discussed next.

Whenever an operation b token appears at the head of the queue for register i, with $i \in D(b)$, a virtual operation is set up, immediately transferring the contents of register i into the domain buffer for this operation. We know that the contents of register i are valid, since i appeared at the head of the queue. The token is then removed from the queue, and similar buffering can occur for other operations. When its domain buffers are filled, a virtual operation can begin. If a range token is at the head of the queue, the operation can be issued; but the token is not removed, and further tokens cannot be examined until this operation has completed. The advantage of this modification is that no interchanging of tokens is necessary. Unfortunately, to be completely competitive with forwarding, the ability to do range buffering must be reintroduced. This has the effect of again multiplying the number of queues, since there would be one queue for each range buffer. Figure 14 illustrates the second queueing technique.

Aside from implementation details, the difference between the schemes discussed in this section and the forwarding schemes discussed in the previous section is mainly one of viewpoints. On the one hand, we view the control of sequencing as being distributed among the virtual function units, and on the other hand as being present in a "global" control unit.

THE CASE FOR DECISIONS

Let us now consider what happens if decisions are allowed in the window. Obviously this means that less instruction-fetching has to be done in the case where a decision causes looping back to another instruction in the window. This is one very practical reason for including decisions in the window.

We now consider what it means to perform "look-ahead" when decisions are involved. If c is a decision, then it is clear that the execution of c must be deferred if there are any operations b with $R(b) \cap D(c) \neq \emptyset$ which are either pending or in execution.

Since a decision affects the flow of control in sequential execution, look-ahead past a decision is normally limited. One possibility is that look-ahead could proceed through

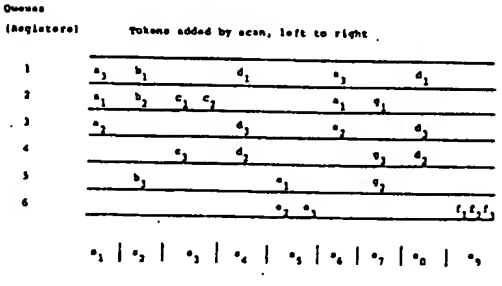


FIGURE 14. The second queueing scheme applied to Example I.

both alternatives of a decision in parallel, and when the decision is finally complete, the results of the operations in the proper alternative would be kept, and the other results destroyed. The control in this case would be extremely complicated, and extra function units would be required to do the "parallel" look-ahead at the same speed as the "serial" look-ahead. Also, if any alternative itself contains a decision, the problem grows rapidly out of proportion. We assume that this type of look-ahead is not used, even though we acknowledge the fact that it may result in some speed-up. We make a similar assumption for any scheme which "guesses" one of the alternatives and conditionally executes the corresponding operations. We claim the additional hardware costs that would be incurred in all these cases are not justifiable.

Having made these assumptions, what is left to be considered? First, it is possible that some operation will be executed regardless of which particular alternative of a decision occurs. Furthermore, this operation may be such that its operands are available before the decision is executed. Hence, such an operation may be "pulled," or "percolated" through the decision and executed before, or concurrently with, the decision. We show in [Ke] that such operations, if they are not decisions, can be detected and percolated by preprocessing the program. This is illustrated in Figure 15. If percolation is done prior to execution, then the task need not be performed by the operation-issuing unit. In fact, we see no practical way of handling this other than by preprocessing.

If the operation to be percolated is a decision, the problem is greater because of the possibility of interchanging, or concurrently executing, two or more decisions. Matters then become complicated because we have a total number of alternatives that is the product of all the individual alternatives. We have not yet found a way to handle all of these conveniently by a look-ahead mechanism. Hence, we make the following assumption.

Assumption 4: No two decisions are executed concurrently, and each is executed in the order specified in the original program.

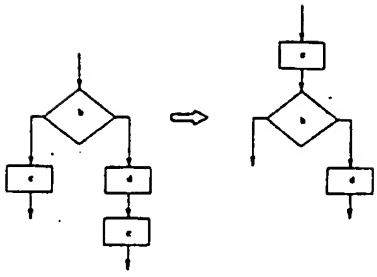


FIGURE 15. Illustrating "percolation"—no operations are conflicting.

Techniques that account for concurrent execution of decisions have been described [Da], but the feature that excludes those techniques from practical consideration here is that parallelism is explicitly specified to a machine capable of interpreting the specification, rather than it being implicitly specified, as in the case of a look-ahead processor.

A subtle point that occurs when decisions are permitted, even with the restrictions stated above, is that previously-issued operations can be executing or pending while a decision is executing. This means that the execution of operations from two different iterations of a loop may overlap in time. We are then led to the observation that, contrary to the case in which there are no decisions, *no finite control can be optimal* when decisions are permitted. A formal proof of this fact is given in [Ke], so here we present an informal and more intuitive version. Observe the following program:

s_1: $1 \leftarrow F_a(1, 1)$
s_2: $2 \leftarrow F_b(2, 2)$
s_3: If $G_c(2)$ then go to s_1

Although this program may seem trivial, or somewhat contrived, it abstracts a situation that may occur in more complex examples. Presumably, registers 1 and 2 have been suitably initialized. Suppose the execution time of the operation corresponding to s_1 will always take nominally longer than s_2 and s_3 combined. An optimal look-ahead control will note that whenever s_1 is executed and

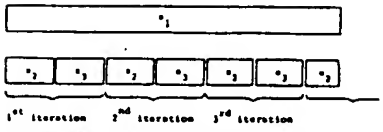


FIGURE 16. Illustrating the execution of a program in which a large number of pending executions of an operation need to be remembered by the control.

the condition $G_s(2)$ is satisfied, resulting in a transfer to a_1, then a_1 and the sequence a_2 followed by a_3 can be processed concurrently. Suppose a_3 is executed with the outcome being a transfer to a_1, but the previous operation generated by a_1 has not yet completed. Since a_1 uses register 1, the next operation to be generated by a_1 cannot begin, although a_2 can begin. For maximal parallelism, a_2 must be issued, and therefore the control must remember the pending execution of a_1. Now if the first a_1 is sufficiently slow, the second a_2, and then a_3 will both be completely executed before the first a_1 completes. If the test $G_s(2)$ is again satisfied, then control must remember the pending execution of two copies of a_1, and so on. Even if some copies of a_1 do completely execute as time goes on, the control *may* have to remember that arbitrarily many copies of a_1 are pending. But no finite control can count to an arbitrarily high number, therefore no finite control is optimal. Figure 16 shows part of a timing diagram in which the pending execution of a large number of a_1 have to be remembered by the control.

By using an argument similar to that in the preceding paragraph, it can be demonstrated that no control which uses only a fixed number of counters, even if these are allowed to be unbounded, can be optimal [Ke]. However, it is shown in the same reference that if queues of unbounded length are allowed, then under the restriction on decisions stated earlier, an optimal look-ahead can be constructed.

The fact that an optimal control can be constructed with unbounded queues may not appear to be of much consolation to the system designer. However, the construction technique does offer valuable conceptual information on the organization of a finite control.

Our discussion shows that when using look-ahead schemes, the lengths of queues are *artificially* bounded by implementation considerations. That is, for some programs it is always possible to get a greater speedup by adding more control states. Tradeoffs involving lengths can be determined by simulation of typical instruction streams.

SCHEDULING

Thus far we have been concerned only with the detection of parallelism as it occurs in *issuing* operations. If each issued operation can be immediately assigned to a *real* function unit, we would then have the maximal degree of parallelism allowable by the parallelism-detection mechanism.

The insurance that there are enough function units available for optimality would probably result in idle units a large percentage of the run-time, as it is unlikely that all types of functions will be used with constant frequency. However, if there are fewer real function units than generated operations, then some choices must be made concerning the order of execution of the operations. This problem of *scheduling* operations has an arbitrary solution insofar as logical dependencies among instructions are concerned, but it can have an effect on the time required to execute a program. This is because the order of execution of operations may determine the order in which subsequent operations can be issued, and thereby determine whether certain function units will be idle. Some subtleties of scheduling are discussed in [G1, G2].

One solution to the scheduling problem is to have the compiler order its code so an efficient execution is obtained when the code is executed, according to scheduling on a fixed basis, say first-in-first-out. In other words, the first virtual unit in a specific ordering whose operands are available is the next to be assigned to a real function unit. Although such ordering by a compiler might be possible, there are a number of drawbacks:

1) such a procedure necessarily assumes

a static window, and therefore may be of questionable validity;

2) very few cases exist for which fast scheduling algorithms are known. Furthermore, recent work [U] has indicated that scheduling algorithms are generally prohibitively time-consuming; and
3) the code produced is likely to be highly machine-dependent. This may be undesirable.

Thus, it appears that preprocessing by the compiler is best limited to fast heuristics that are very likely to increase concurrency. Some bounds on the worst case for such heuristics in the instance of identical function units are derived in [G1]; these appear encouraging because they indicate that trivial scheduling schemes can produce executions that differ from the optimal by at most a small multiplicative factor. However, similar results have not been shown for nonidentical function units. Derivation of bounds in this more general case is a problem for future research.

We may ask similar questions about dynamic scheduling within the window itself. Certainly if fast compiler techniques for scheduling are difficult to find, then the same is true for fast and sufficiently inexpensive dynamically-executable scheduling techniques. However, there is a subtle distinction. Scheduling within a machine may be made to depend on the precise knowledge of which resources are available at any instant of time, and the strategy may be varied accordingly. The effectiveness of dynamic scheduling remains open for future investigation.

ALGEBRAIC IDENTITIES

This and the following section survey other possible enhancements to look-ahead mechanisms. We make no attempts to present optimal methods when these enhancements are allowed, as the problem of doing so appears intractible.

According to Assumption 2, (page 180) we have been assuming that no nontrivial relations hold between different operations. Now we consider the relaxation of this constraint.

First, let us consider the case in which two or more functions with different names are in fact equivalent. This may be useful in allowing the programmer to bypass any built-in scheduling mechanism, enabling him to preplan the schedule for greater parallelism. For example, if there are three adder units, then the programmer may use a different function code for adder 1, adder 2, or adder 3. The specification of a particular adder would indicate that if the adder is busy, the operation is not to be issued, even if another adder is available. It is not difficult to provide examples wherein this form of "balking" results in reduced execution time. A special function code might be used to indicate that the programmer doesn't care which unit is used, and the choice can be made arbitrarily and dynamically.

Another type of relation among operations is associativity and/or commutativity of, for example, addition or multiplication. Although associativity does not hold for either of these operations in the domain of floating-point numbers, it may be assumed, with the knowledge that the results of a series of such operations may not be precisely determined. It has been observed, e.g., in [KMC], that associativity and commutativity allow possible speedups in the execution of programs for arithmetic expressions without introducing any additional operations. On the other hand, the assumption that multiplication distributes over addition can be used to effect a speedup, but additional operations must generally be introduced. Therefore, in a case where the number of real function units is a limiting factor, this speedup may not materialize. This indicates that a rather machine-dependent compiler may be necessary to take full advantage of the available resources. Alternatively, it may be possible to have the control dynamically decide whether to apply an algebraic identity, depending on the availability of real function units. To the author's knowledge, no efficient techniques for accomplishing this are known.

With regard to the type of dynamic decisions mentioned here, however, one tech-

nique we would like to suggest is the use of more complex instructions. For example, a single instruction might specify "add the contents of the next 5 registers listed." If there is no implied ordering of the operands, then associativity and commutativity are being assumed. The contents of some registers might not yet have been computed, but a set of adder units could go to work on those that have, adding pairs of operands as they become available. This gives a more flexible ordering than is possible with the technique of requiring a predefined sequence of additions.

OTHER CONSIDERATIONS

We now mention some other factors that relate to the effectiveness of a look-ahead processor. First, there is a generalization of forwarding and buffering to allow arbitrary *register renaming*.

It may be observed that there is nothing particularly sacred about the register in which a value is stored. The index of a register is simply a *name* by which that value can be accessed when fetching is necessary. Thus, the physical registers used are really arbitrary, as long as we have a way of recalling the value associated with a particular name. This indicates that it is possible to reduce register conflicts by dynamically renaming registers.

For example, suppose an instruction specifies that a certain value is stored in register i. Suppose, also, that some instruction which appears several instructions later in the stream also specifies storage into i. The latter instruction cannot normally be executed in advance because instructions between it and the former instruction may reference register i. However, the following scheme may be used to allow intermediate instructions to proceed in parallel with subsequent instructions, provided that no further conflicts arise.

Suppose we associate a unique index with each physical register, and consider the indices specified in instructions as *names* for these registers. In general, there are to be more registers than names. In addition, we assume that there is a *mapping table* which gives for each name the index of the physical register with which the name is currently associated. When an operation involving register i is issued, i is translated into the index of the physical register currently assigned to i. Henceforth, the operation addresses this register through its physical index.

When register i is specified as the name of a range register, a new physical register is assigned, and the mapping table is updated to reflect this. The former physical register is now inaccessible to future instructions. So the control knows when an inaccessible physical register is to be reassigned, a count similar to W_i in the section on "Elementary Schemes" must be associated with each physical register, indicating how many operations have been issued that will reference its value. As each reference occurs, the count is decremented, and when it reaches zero, the register is available for reassignment. Observe that with a general renaming scheme, it is unnecessary to have more than one register name in order to make use of multiple registers. Thus, even a single-accumulator architecture will suffice. Stone [S, E] has taken this a step further by suggesting the use of a renaming scheme in conjunction with an "addressless" pushdown stack machine. With increased use of cache memory organizations, which remap registers in any case, the renaming scheme becomes increasingly attractive.

The technique of *multiple instruction streams* may be used to increase the efficiency of a look-ahead processor. Since it is not likely that a typical instruction stream can make use of all of a large collection of function units and/or registers, we may allow several different operation-issuing units to issue operations from different instruction streams to a pool of function units. Each stream is associated with a separate instruction pointer and decoding unit. This has been suggested in [AFR, FPS], for example. Multistream organizations without interactive sharing are presented in [H, Th].

The principal usefulness of the multistream technique is based on the assumption that several streams are likely to make more uniform use of the pool of function units than one stream. As with several other tech-

niques which we have discussed, the increased complexity of the control may outweigh the gain in efficiency. This is due in part to the requirement that there be mechanisms for resolving conflicting requests for function units. Also, when accompanied by a register renaming scheme, "deadlocks" can result, as mentioned in [Co].

Another concept which relates to those we have discussed is that of "pipelining" [HT, Wat]. By this term we mean the execution of a sequence of similar operations in a way that allows concurrent computation of different suboperations of more than one operation. For example, a floating-point addition typically consists of three phases: alignment, fraction addition, and normalization. Suppose each phase is considered a distinct suboperation, performed by a distinct subfunction unit. Then at most one subunit will be used at a time for processing any given operation, and hence this unit can be used to process suboperations of other operations. We can consider the three subfunction units as forming a "pipe" through which operations flow. Assuming each suboperation requires the same amount of time, if there are n distinct suboperations, then the net time to execute a large number of operations is roughly $1/n$ of the time required for sequential execution.

Pipelining relates to our discussion of look-ahead in two ways. The first is that look-ahead itself may be considered a form of pipelining in which the operations can be highly-dissimilar, provided operations are scheduled on virtual function units on a first-in-first-out basis. Second, a function unit may itself be implemented as a pipeline that is fed by the corresponding virtual function units.

An interesting coupling of pipeline and look-ahead processing concepts is discussed in [Cr]. Here the registers are capable of holding vectors of operands. Look-ahead control can be organized as we have discussed earlier. However, by appropriate synchronization, buffering can be achieved with single component buffers, instead of by the obvious approach of buffering entire vectors.

The techniques of register renaming, pipelining, and multiple-streams have prompted some authors to consider more radical machine organizations [DM, MC, MT, Ro]. This has led to the *data flow* programming concept, in which a program is specified as a graph, similar to the precedence graph, rather than as a sequence of instructions. The idea is to eliminate the "intermediate" sequential program from the machine-interpretation phase of problem solution. The concept apparently originated theoretically in [KM1].

As our final consideration, we should mention that any potential increase in performance can be shattered if the instruction stream is subject to frequent interrupts. The reason for this, of course, is that when an interrupt occurs, if the interrupt routine is to be able to use programmable registers, then all operations in progress must be complete before the register contents can be saved and the instructions in the interrupt routine can be processed. This grows more complicated if the interrupts are due to some aspect of the execution of the operations themselves, such as the occurrence of an arithmetic overflow. The latter consideration has led to the notion of "imprecise interrupt" [AST]. This means that interrupts which cannot be precisely associated with any one instruction are allowed to occur, but the general vicinity of the instruction is known. In the machine described in [AST], for example, this feature can be turned off and instructions processed serially.

The interrupt problem can be alleviated in part by using a flexible register renaming scheme, such as that described earlier. However, it is probably a better idea to decrease the frequency of interrupts, handling them on a separate "peripheral" processor if possible. For a comparison of approaches, see [AST, Th].

ACKNOWLEDGMENT

The author wishes to thank Arch Davis and Leonard Vanek for providing comments on the manuscript. This work was sponsored by NSF Grants GJ-30126 and GJ-42627.

REFERENCES

[AFIR] Aschenbrenner, R. A.; Flynn, M. J.; and Robinson, G. A. "Intrinsic multiprocessing," *Proc. AFIPS, 1967 Spring Jt. Computer Conf.*, Vol. 30, AFIPS Press, Montvale, N. J., 1967, pp. 81–86.

[AGU] Aho, A. V.; Garey, M. R.; and Ullman, J. D. "The transitive reduction of a directed graph," *SIAM J. Computers*, 1, 2 (June 1972), 131–137.

[AST] Anderson, D. W.; Sparacio, F. J.; and Tomasulo, R. M. "The IBM System/360 model 91: machine philosophy and instruction handling," *IBM J. R&D*, 11, 1 (Jan. 1967), 8–24.

[B] Bernstein, A. J. "Program analysis for parallel processing," *IEEE Trans. Electronic Computers*, EC-15, (Oct. 1966), 757–762.

[Co] Coffman, E. G. "A formal microprogram model of parallelism and register sharing," *Symposium on Computers and Automata*, Polytechnic Institute of Brooklyn, New York, (April 1971), 215–223.

[Cr] Cray Research, Inc. *Cray-1 Preliminary Reference Manual* (Draft), (Feb. 1975).

[Da] Davis, E. W. "Concurrent processing of conditional jump trees," *IEEE Compcon '72*, IEEE, New York, (Sept. 1972), 279–281.

[De] Dennis, J. B. "Modular, asynchronous control structures for a high performance processor," *ACM Conf. Record, Project MAC Conf. on Concurrent Systems and Parallel Computation*, (June 1970), 55–80.

[DM] Dennis, J. B.; and Misunas, D. P. "A preliminary architecture for a basic data-flow processor," MIT Project MAC Computation Structures Group Memo. 102 (August 1974).

[E] Elspas, B., et al. "Investigation of propagation-limited computer networks," Stanford Research Institute Report AFCRL-64-376 (III). AD 637 769 (June 1966).

[FPS] Flynn, M. J.; Podvin, A.; and Shimizu, K. "A multiple instruction stream processor with shared resources," in *Parallel processor systems, technologies, and applications*, Spartan Books, Washington, D.C., 1970. pp. 251–286.

[G1] Graham, R. L. "Bounds on multiprocessing timing anomalies," *SIAM J. Appl. Math.*, 17, 2, (March 1969), 416–429.

[G2] Graham, R. L. "Bounds on multiprocessing anomalies and related packing algorithms," *Proc. AFIPS 1972 Spring Jt. Computer Conf.*, Vol. 40, AFIPS Press Montvale, N. J., 1972, pp. 205–217.

[H] Harper, S. D. "Automatic parallel processing," *Proc. Computing and Data Processing Society of Canada*, Second Conference, (June 1900), 321–331.

[HT] Hentz, R. G.; and Tate, G. P. "Control Data Star-100 processor design," *IFEE Proc. Compcon '72*, IEEE, New York, (Sept. 1972), 1–4.

[KM1] Karp, R. M.; and Miller, R. E. "Properties of a model for parallel computations: determinacy, termination, queueing," *SIAM J. Appl. Math.*, 14, 6 (Nov. 1966), 1390–1411.

[KM2] Karp, R. M.; and Miller, R. E. "Parallel program schemata," *J. Computer & System Sciences* 3, 2 (May 1969), 147–195.

[Ke] Keller, R. M. "Parallel program schemata and maximal parallelism," *J. ACM* 20, 3 (July 1973) 514–537; and *J. ACM* 20, 4 (Oct. 1973), 696–710.

[KMC] Kuck, D. J.; Muraoka, Y.; and Chen, S.-C. "On the number of operations simultaneously executable in Fortran-like programs and their resulting speed-up," *IEEE Trans. Computers*, C-21, 12 (Dec. 1972), 1293–1309.

[MC] Miller, R. E.; and Cocke, J. "Configurable computers: a new class of general-purpose machines," in *International symposium on theoretical programming*, Ershov and Nepomniaschy (Eds.), Springer Verlag, New York, 1974, pp. 285–298.

[MT] Morris, D.; and Treleaven, P. C. "A stream processing network," *Sigplan Notices*, 10, 3, (March 1975), 107–112.

[Ro] Rohrbacher, D. L. *Advanced computer organization study*, Rome Air Development Corp., Tech. Report. RADC-TR-66-7 (2 vols.) AD 631 870, and 631 871 (April 1966).

[S] Stone, H. S. "A pipeline push-down stack computer," in *Parallel processor systems, technologies, and applications*, Spartan Books, Washington, D.C., 1970, pp. 235–249.

[Th] Thornton, J. E. *Design of a computer system: the Control Data 6600*, Scott, Foresman, and Company, 1970.

[To] Tomasulo, R. M. "An efficient algorithm for exploiting multiple arithmetic units," *IBM J. R&D*, 11, 1 (Jan. 1967), 25–33.

[U] Ullman, J. D. "Polynomial complete scheduling problems," *Operating Systems Review*, 7, 4, (Oct. 1973), 96–101.

[War] Warshall, S. "A theorem on Boolean matrices," *J. ACM*, 9, 1, (Jan. 1962), 11–12.

[Wat] Watson, W. J. "The Texas Instruments Advanced Scientific Computer," *IEEE Proc. Compcon '72*, IEEE, New York, (Sept. 1972), 291–293.

SUPPLEMENTARY REFERENCES

[1] Allard, R. W.; Wolf, K. A.; and Zemlin, R. A. "Some effects of the 6600 computer on language structures," *Comm. ACM*, 7, 2, (Feb. 1964), 112–119.

[2] Buccholz, W., [Ed.] *Planning a computer system*. McGraw-Hill, New York, 1962.

[3] Chen, T. C. "The overlap design of the IPM System/300 model 92 central processing unit," *Proc. AFIPS 1964 Spring Jt. Computer Conf.*, Vol. 25 AFIPS Press, Montvale, N. J., 1964, pp. 73–80.

[4] Flynn, M. J. "Some computer organiza-

tions and their effectiveness," *IEEE Trans. Computers*, C-21, 9 (Sept. 1972), 948–960.

[5] FOSTER, C. C.; AND RISEMAN, E. M. "Percolation of code to enhance parallel dispatching and execution," *IEEE Trans. Computers*, C-21, 12 (Dec. 1972), 1411–1415.

[6] FRANKOVICH, J. M.; AND PETERSON, H. P. "A functional description of the Lincoln TX-2 computer," *Proc. Western Jt. Computer Conf.*" (Feb. 1957), 146–155.

[7] GRAHAM, W. R. "The parallel and the pipeline computers," *Datamation*, 16, 4 (April 1970), 68–71.

[8] IBBETT, R. N. "The MU5 instruction pipeline," *Computer J.*, 15, 1 (Feb. 1972), 42–47.

[9] LOCRIPPO, L. "Renaming in program schemas," *Proc. IEEE 13th Annual Symposium on Switching and Automata Theory*, (Oct. 1972), 67–70.

[10] MILLER, E. F. "A multiple-stream registerless shared-resource processor," *IEEE Trans. Computers* C-23, 3 (March 1974) 277–285.

[11] RIEGEL, E. W. *Parallelism exposure and exploitation in digital computing systems*. Tech. Report TR-09-4, Burroughs Corp., Defense, Space, and Special Systems Group, 1969.

[12] RISEMAN, E. M.; AND FOSTER, C. C. "The inhibition of parallelism by conditional jumps," *IEEE Trans. Computers*, C-21, 12 (Dec. 1972), 1405–1410.

[13] SHEMER, J. E.; AND GUPTA, S. C. "A simplified analysis of processor look-ahead and simultaneous operation of a multiple-module main memory," *IEEE Trans. Computers*, C-18, 1 (Jan. 1969), 64–71.